NRL Release Number 06-1226-2883

Formal Specification and Verification of Data Separation in a Separation Kernel for an Embedded System

Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean
Naval Research Laboratory, Washington, DC 20375
{heitmeyer, archer, leonard, mclean}@itd.nrl.navy.mil

Abstract

Although many algorithms, hardware designs, and security protocols have been formally verified, formal verification of the security of software is still rare. This is due in large part to the large size of software, which results in huge costs for verification. This paper describes a novel and practical approach to formally establishing the security of code. The approach begins with a well-defined set of security properties and, based on the properties, constructs a compact security model containing only information needed to reason about the properties. Our approach was formulated to provide evidence for a Common Criteria evaluation of an embedded software system which uses a separation kernel to enforce data separation. The paper describes 1) our approach to verifying the kernel code and 2) the artifacts used in the evaluation: a Top Level Specification (TLS) of the kernel behavior, a formal definition of data separation, a mechanized proof that the TLS enforces data separation, code annotated with pre- and postconditions and partitioned into three categories, and a formal demonstration that each category of code enforces data separation. Also presented is the formal argument that the code satisfies the TLS.

Categories and Subject Descriptors: D.2.4 [Software]: Software Engineering

General Terms: security, verification, languages, theory

Keywords: formal model, formal specification, theorem proving, separation kernel, code verification

1. Introduction

A critical objective of many military systems is to protect the confidentiality and integrity of sensitive information. Preventing unauthorized disclosure and modification of information is of enormous importance in military systems, since violations can jeopardize national security. Compelling evidence is required therefore that military systems satisfy their security requirements.

A promising approach to demonstrating the security of code is formal verification, which has been successfully applied to algorithms, such as floating point division [26] and clock synchronization [31], and security protocols such as cryptographic protocols [24, 20]. However, most past efforts to verify security-critical software have been extremely expensive. One reason is that these efforts often built security models containing too much detail (see, for example, [11]) or tried to prove too many properties (see, for example, [36]). The result was that model building and property proving became prohibitively expensive.

A challenging problem therefore is how to make the verification of security-critical code affordable. This paper describes an approach to verifying the security of software that is both novel and practical. This approach was formulated in preparation for a Common Criteria evaluation of the security of a software-based embedded device called ED (Embedded Device). For the ED application, satisfying the Common Criteria required a formal proof of correspondence between a formal specification of ED's security functions and its required security properties *and* a demonstration that the code implementing ED satisfied the formal specification. ED, which processes data stored in different partitions of memory, is required to enforce a critical security property called *data separation*; for example, ED must ensure that data in one memory partition neither influences nor is influenced by data in another partition. To ensure that data separation is not violated, or if it is violated an exception occurs, the ED architecture includes a separation kernel [33], a tamper-proof, non-bypassable program that mediates every access to memory.

The task of our group was to provide evidence to the certifying authority that the ED separation kernel enforces data separation. The kernel code, which consists of over 3000 lines of C and assembly code, was annotated with pre-and postconditions in the style of Hoare and Floyd. To provide evidence that ED enforces data separation, we produced a Top Level Specification (TLS) of the separation-relevant behavior of the kernel, a formal statement of data separation, and a mechanized formal proof that the TLS satisfies data separation. Then, the annotated code was partitioned into three categories, each requiring a different proof strategy. Finally, the formal correspondence between the annotated code and the TLS was established for each category of code. Recently, five artifacts—the TLS, the formal statement of data separation, proofs that the TLS satisfies data separation, the organization of the annotated code into the three categories, and the documents showing correspondence of each category of code with the TLS—were presented along with the annotated code as evidence in a Common Criteria evaluation of ED.

This paper summarizes the process we followed in producing evidence for the Common Criteria evaluation of ED's separation kernel, describes each artifact developed during this process, and summarizes both the formal state machine model that underlies the

This paper is authored by employees of the United States Government and is in the public domain.
CCS'06, October 30–November 3, 2006, Alexandria, Virginia, USA.
ACM 1-59593-518-5/06/0010.

TLS and the formal argument justifying our approach to establishing code conformance with the TLS. The paper makes two technical contributions. First, it describes a novel technique for partitioning the code into three different categories—namely, Event, Trusted, and Other Code—and for reasoning about the security of each category. Second, it describes a method for demonstrating the security of the code that is both original and practical. While the method combines a number of well-known techniques for specifying and reasoning about security—e.g., a state machine model represented both formally and in natural language, mechanized reasoning using PVS [34], and a demonstration of correspondence between the TLS and the annotated source code—which techniques to apply, how to apply them, and how to combine them was far from obvious and required significant discussion during the course of the project. Along the way, many alternative approaches and techniques were considered, and several were discarded. In our view, both the technique for partitioning the code and the method we formulated for proving that the code is secure should prove useful in future efforts to verify the security of software.

The paper is organized as follows. Section 2 reviews the notion of a separation kernel, summarizes the requirements of a Common Criteria evaluation, and presents some details of ED. Section 3 describes the process we followed to demonstrate data separation and describes the five artifacts that the process produced, including the three categories of code and how we proved that each category of code is secure. Section 4 presents the formal argument for demonstrating code conformance. Sections 5 and 6 discuss some lessons learned and describe topics requiring more research, i.e., the need for more powerful tool support. Section 7 describes related work. Finally, Section 8 presents some conclusions.

2. Background

2.1 Separation Kernel

A *separation kernel* [33] mimics the separation of a system into a set of independent virtual machines by dividing the memory into partitions and restricting the flow of information between those partitions. Separation kernels are being developed for military applications requiring Multiple Independent Levels of Security (MILS) by commercial companies such as Wind River Systems, Green Hills Software, and LynuxWorks [4]. In a MILS environment, a separation kernel acts as a reference monitor [6], i.e., is non-bypassable, evaluatable, always invoked, and tamper proof.

2.2 Common Criteria

Seven international organizations established the Common Criteria to provide a single basis for evaluating the security of information technology products [3]. Associated with the Common Criteria are seven Evaluation Assurance Levels. EAL7, the highest assurance level, requires a formal specification of a product's security functions and its security model, and formal proof of correspondence between the two.

2.3 Embedded Device (ED)

The device of interest in this paper, ED, processes data in an embedded system. While at any given time the data stored and processed by ED in one memory partition is classified at a single security level, ED may later reconfigure that partition to store and process data at a different security level. Because it stores and processes data classified at different security levels, security violations by ED could cause significant damage. To prevent violations of data separation, e.g., the "leaking" of data from one memory partition to another, the ED design uses a separation kernel to mediate access to memory. By mediating every access, the kernel ensures that every memory access is authorized and that every transfer of data from one ED memory location to another is authorized. Any attempted memory access by ED that is unauthorized will cause an exception. Section 3.3 describes how TAME [8, 7], an interface to SRI's theorem prover PVS [34], was used to support the Common Criteria evaluation of ED's separation kernel.

3. Code Verification Process

The process followed in constructing the five ED artifacts consists of five steps. The process described below is an idealization of the actual process since, in any real-world process, one frequently returns to a former step to make corrections and add missing information. However, the sequence of steps that follows is a logical order for producing the various artifacts.

1. Formulate a Top Level Specification (TLS) of the kernel as a state machine model, using the style introduced in [21, 23].

2. Formally express the data separation property in terms of the inputs, state variables, and transitions defined in the state machine model that underlies the TLS.

3. Translate the TLS and the data separation property into the language of a mechanical prover, and prove formally that the TLS satisfies the data separation property.

4. Given a source code implementation of the kernel annotated with pre- and postconditions, partition the code into Event, Other, and Trusted Code, where, informally, Event Code is code corresponding to an event in the TLS that touches a Memory Area of Interest (defined below), Trusted Code is code that touches a Memory Area of Interest but is not Event Code, and Other Code is neither Event Code nor Trusted Code. Section 3.4 provides precise definitions of the three different code categories.

5. Demonstrate that the Event Code does not violate separation by constructing 1) a mapping from the Event Code to the TLS events and from the code states to the states in the TLS, and 2) a mapping from pre- and postconditions of the TLS events to pre- and postconditions that annotate the corresponding Event Code. Demonstrate separately that Trusted Code and Other Code do not violate data separation.

3.1 Top Level Specification

Major goals of the Top Level Specification (TLS) are to provide a precise, yet understandable description of the required external behavior of ED's separation kernel and to make explicit the assumptions on which the specification is based. To achieve this, the TLS represents the kernel as a state machine model using precise natural language. Such a natural language description was introduced in 1984 to describe the behavior of a secure military message system (MMS) [21, 23]. The advantage of natural language is that it enables stakeholders from differing backgrounds and with different objectives—the project manager, software developers, evaluators, and formal methods team—to communicate precisely about the required kernel behavior and helps ensure that misunderstandings are weeded out and issues resolved early in the verification process. Another goal of the TLS is to provide a formal context and precise vocabulary for defining data separation.

Like the secure MMS model, the state machine representing the behavior of the ED kernel is defined in terms of an input alphabet, a set of states, an initial state, and a transform relation describing the allowed state transitions. The input alphabet contains internal and external events, where an *internal event* can cause the kernel to invoke some process and an *external event* is performed by an external host. An example of an internal event is an event instructing ED to copy data from an input buffer associated with memory partition i to a data area in partition i. An example of an external event is the event occurring when an external host writes data into an input buffer assigned to partition i. The transform (also called the *next-state relation*) is defined on triples consisting of an event in the input alphabet, the current state, and the new state. Provided below are excerpts from the TLS as well as an example internal event. This event, `Copy_Buf1In_Data1In_`i, copies data from an input buffer for partition i into a data area in partition i.

Partitions, State Variables, Events and States. We assume the existence of $n \geq 1$ dedicated memory partitions and a single shared memory area. We also assume the existence of the following sets:

- V is a union of types, where each type is a non-empty set of values.
- R is a set of state variable names. For all r in R, $\mathrm{TY}(r) \subseteq V$ is the set of possible values of state variable r. $\mathcal{M}$ is a union of N non-overlapping memory areas, each represented by a state variable.
- $H = P \cup E$ is a set of M events, where each event is either an internal event in P or an external event in E.

A *system state* is a function mapping each state variable name r in R to a value. Formally, for all $r \in R$: $s(r) \in \mathrm{TY}(r)$.

Memory Areas. The N memory areas contain $N - 1$ Memory Areas of Interest, where $N - 1 = mn$, and m is the number of Memory Areas of Interest per partition. Informally, a Memory Area of Interest (MAI) is a memory area containing data whose leakage would violate data separation. The m MAIs for a partition i, $1 \leq i \leq n$, include partition i's input and output buffers and k data areas where data in partition i is stored and processed. The Nth memory area, called G, contains all programs and data not residing in an MAI and is the single shared memory area. The set $\mathcal{M}$ of all memory areas is defined as the union $A \cup \{G\}$, where $A = \{A_{i,j} \mid 1 \leq i \leq n \wedge\ 1 \leq j \leq m\}$ contains the mn MAIs. For all i, $1 \leq i \leq n$, $A_i = \{A_{i,j} \mid 1 \leq j \leq m\}$ is the set of memory areas for partition i. To guarantee that the memory areas of $\mathcal{M}$ are non-overlapping, the memory areas are required to be pairwise disjoint.

State Variables. The set of state variables[1] contained in R are

- a partition id c,
- the N memory areas in $\mathcal{M}$, and
- a set of n sanitization vectors $\mathcal{W}_{\mathrm{D}}[1], \ldots, \mathcal{W}_{\mathrm{D}}[n]$, each vector containing k elements.

The partition id c is 0 if no data processing in any partition is in progress, and i, $1 \leq i \leq n$, if data processing is in progress in partition i. (Data processing can occur in only one partition at a time.) For $1 \leq j \leq k$, the boolean value of the jth element $\mathcal{W}_{\mathrm{D}}^{j}[i]$ of the sanitization vector for partition i is *true* if the jth memory area of the ith partition has been sanitized and *false* otherwise. A sanitized memory area is modeled as having the value 0.

Events. The set of internal events $P \subset H$ is the union of n sets, $P_1, \ldots, P_n$, of *partition events*, one set for each partition i, and a singleton set Q; thus P is defined by $P = [\cup_{i=1}^{n} P_i] \cup Q$. Processing occurs on partition i when a sequence of events from P_i is processed. The sole member of Q is the event `Other_NonParProc`, an abstract internal event representing all internal events which invoke data processing in the shared message area G. One example of such an event is `Assign_Val`, which causes some value to be stored in G. The set of external events $E \subset H$ is defined by $E = E^{\mathrm{In}} \cup E^{\mathrm{Out}} \cup \{\texttt{Ext\_Ev\_Other}\}$, where $E^{\mathrm{In}} = \cup_{i=1}^{n} E_i^{\mathrm{In}}$ and $E^{\mathrm{Out}} = \cup_{i=1}^{n} E_i^{\mathrm{Out}}$. E_i^{In} is the set of external events writing into or clearing the input buffers assigned to partition i, and E_i^{Out} is the set of external events reading from or clearing the output buffers assigned to partition i. The event `Ext_Ev_Other` represents all other external events.

Partition Functions. Operations on data in partition i, for example, an operation copying data from one MAI in partition i to another MAI in i, are called 'partition functions.' For all i, $1 \leq i \leq n$, and for each internal event e in P_i, there exists a *partition function* Γ_e associated with e. Each function Γ_e computes a value stored in an MAI in A. For all $e \in P_i$, Γ_e has the signature $\Gamma_e :\ \mathrm{TY}(a_1) \rightarrow \mathrm{TY}(a_2)$, where a_1 and a_2 are MAIs in A_i. Thus, each function Γ_e, where e is an internal event in P_i, takes a single argument, the value stored in some MAI a_1, and uses that argument to compute a value to be stored in MAI a_2.

Access Control Matrix. Associated with the M events and the N memory areas is an M by N access control matrix AM, which indicates the read and write access that each internal event e in P (and its associated process) and each external event e in H has for each memory area a in $\mathcal{M}$. Each entry in the matrix is either `null` meaning no access, `R` for read access, `W` for write access, or `RW` for both read and write access. The left-most column of AM lists the events in H, and the headings of the remaining columns list the N memory areas in $\mathcal{M}$ as well as G.

For all i, j, $1 \leq i, j \leq n$, $i \neq j$, an event associated with partition i has `null` access to an MAI associated with partition j or to G; similarly, an event associated with j has `null` access to an MAI associated with i or to G. Moreover, the single event that invokes non-partition processing, namely, `Other_NonParProc`, has `R` and `W` access for G and access `null` for all other memory areas, i.e., the MAIs. Finally, the external events associated with partition i can only write into or read from input and output buffers associated with i.

System. A system is a state machine whose transitions from one state to the next are triggered by events. Formally, a system Σ is a 4-tuple $\Sigma = (H, S, s_0, T)$, where

- H is the set of events,
- S is the set of states,
- s_0 is the initial state, and

---

[1] By convention, state variable names may refer to the values of the variables.

- T is the system transform, a partial function from $H \times S$ into S. T is partial because not all events are 'enabled' to be executed in the current state.

Initial State. In the initial state s_0, the partition id c is 0; for all i, $1 \le i \le n$, the MAIs in A_i are 0; and each element of the sanitization vectors, $\mathcal{W}_{\mathrm{D}}[1] \ldots \mathcal{W}_D[n]$, is *true*. Hence, in the initial state, no processing in any partition is authorized, only a non-partition process is authorized to execute, each element of every sanitization vector has the value *true*, and all MAIs have the value zero.

System Transform. The transform T is defined in terms of a set $\mathcal{R}$ of transform rules, $\mathcal{R} = \{\, \mathcal{R}_e \mid e \in H \,\}$, where each *transform rule* $\mathcal{R}_e$ describes how an event e transforms a current state into a new state. The number of rules is M, one rule for each of the M events in H. No rule requires access privileges other than those defined by the access control matrix AM. The notation s and s' represents the current state and the new state. Given state s and state variable r, r's value in s is denoted by r_s. When an internal or external event e does not affect the value of any state variable r, when the precondition is not satisfied, or when the event e is not enabled, the value of r does not change from state s to state s', and the state variable r retains its current value, i.e., $r_s = r_{s'}$.

To denote that no state variable except those explicitly named changes, we write $\mathrm{NOC}_{\hat{R}}$ (NO Change except to variables in $\hat{R}$), where $\hat{R} \subset R$. This includes the case where the ith element of a sanitization vector changes but no other vector elements change. For example, the postcondition $r_{s'} = x \,\wedge\, \mathrm{NOC}_{\{r\}}$, where $x \in \mathrm{TY}(r)$, is equivalent to $r_{s'} = x \,\wedge\, \forall\, \hat{r} \in R, \hat{r} \neq r : \hat{r}_{s'} = \hat{r}_s$.

Suppose s is a state in S, e is an event in H, and R is the set of state variables. Let $\texttt{pre}_e$ be a state predicate associated with e such that $\texttt{pre}_e$ evaluates to *true* if e has the potential to occur in state s and and *false* otherwise, and let $\texttt{post}_e$ be a predicate associated with e such that $\texttt{post}_e(s, s')$ holds whenever e occurs in state s and s' is a possible poststate of s when event e occurs in state s. Formally, the transform rule $\mathcal{R}_e$ in $\mathcal{R}$ is defined by

$$\mathcal{R}_e : \texttt{pre}_e(s) \;\Rightarrow\; \texttt{post}_e(s, s').$$

Whenever the result state of every event e is deterministic (which is true in the TLS), the assertion $\texttt{post}_e(s, s')$ defines the poststate $s' = T(e, s)$. To make T total on $H \times S$, the complete definition of T is written as

$$T(e,s) = \begin{cases} s' & \text{if } \texttt{pre}_e(s), \text{ where } \texttt{post}_e(s,s') \\ s & \text{otherwise.} \end{cases}$$

In the above definition, $\texttt{pre}_e(s)$ is not satisfied implies that e has no effect—i.e., essentially, did not occur.

Example of a Transform Rule. Consider an internal event, $e = \texttt{Copy\_Bfr1In\_Data1In\_}i$, which invokes a process copying data from partition i's Input Buffer 1, denoted B_i^1, into partition i's Data Area 1, denoted D_i^1. The transform rule for e is denoted $\mathcal{R}_{\texttt{Copy\_Bfr1In\_Data1In\_}i}$. Preconditions for e are (1) the partition id c equals i, and (2) the invoked process must have read access ('R') for partition i's Input Buffer 1 and write access ('W') for Data Area 1 in i. Postconditions for e are that (3) the element for Data Area 1 in i's sanitization vector becomes *false*, (4) a function of the value stored in i's Input Buffer 1 is written into i's Data Area 1, and (5) no other state variable changes. For all i, the rule $\mathcal{R}_e$ for event $e = \texttt{Copy\_Bfr1In\_Data1In\_}i$ is defined by

$\mathcal{R}_{\texttt{Copy\_Bfr1In\_Data1In\_}i}$:

$$\begin{array}{lll}
 & c_s = i \;\wedge & (1)\\
 & \mathrm{AM}[e,a_1] = \texttt{R} \text{ and } \mathrm{AM}[e,a_2] = \texttt{W} & (2)\\
 & \qquad \text{where } a_1 = B_i^1 \text{ and } a_2 = D_i^1 & \\
\Rightarrow & \mathcal{W}^1_{\mathrm{D},s'}[i] = \textit{false} \;\wedge & (3)\\
 & D^1_{i,s'} = \Gamma_e(B^1_{i,s}) \;\wedge & (4)\\
 & \mathrm{NOC}_{\{\mathcal{W}_D^1[i], D_i^1\}} & (5)
\end{array}$$

3.2 Security Policy: Data Separation

To operate securely, ED must enforce data separation. Informally, this means that ED must prevent data in a partition i from influencing or being influenced by 1) data in a partition j, where $i \neq j$, 2) an earlier configuration of partition i, or 3) data stored in G. Thus ED must prevent non-secure data flows. To demonstrate that the TLS enforces data separation, we proved that it satisfies five subproperties, namely, *No-Exfiltration*, *No-Infiltration*, *Temporal Separation*, *Separation of Control*, and *Kernel Integrity*. Below, each subproperty is defined formally using the notation introduced in Section 3.1.

3.2.1 *No-Exfiltration Property*

The No-Exfiltration Property states that data processing in any partition j cannot influence data stored outside the partition. This property is defined in terms of the set A_j (the MAIs of partition j); the entire memory $\mathcal{M}$; the internal events in P_j, which invoke data processing in j; and external events in $E_j^{\mathrm{In}} \cup E_j^{\mathrm{Out}}$, which affect data in j's input and output buffers.

Property 3.1 (No-Exfiltration) *Suppose that states s and s' are in state set S, event e is in H, memory area a is in $\mathcal{M}$, and j is a partition id, $1 \le j \le n$. Suppose further that $s' = T(e,s)$. If e is an event in $P_j \cup E_j^{\mathrm{In}} \cup E_j^{\mathrm{Out}}$ and $a_s \neq a_{s'}$, then a is in A_j.*

3.2.2 *No-Infiltration Property*

The No-Infiltration Property states that data processing in any partition i is not influenced by data outside that partition. It is defined in terms of the set A_i, which contains the MAIs of partition i.

Property 3.2 (No-Infiltration) *Suppose that states s_1, s_2, s_1', and s_2' are in S, event e is in H, and i is a partition id, $1 \le i \le n$. Suppose further that $s_1' = T(e, s_1)$ and $s_2' = T(e, s_2)$. If for all a in A_i: $a_{s_1} = a_{s_2}$, then for all a in A_i: $a_{s_1'} = a_{s_2'}$.*

3.2.3 *Temporal Separation Property*

The objective of this property is to guarantee that the k data areas in any partition i are clear when the system is not processing data in that partition, e.g., from the end of a processing thread in one partition to the start of a new processing thread in the same or a different partition.[2] Satisfying this property implies a second property—i.e., no data (e.g., Top Secret data) in a partition during one configuration of the ith partition can leak into a later configuration (e.g., at the Unclassified level) of the same partition i. The set of states in which the system is not processing data stored in a partition is exactly the set of states in which the partition id c_s is 0. This fact can be used in stating the Temporal Separation Property.

---

[2] The proof of this property depends on a constraint imposed by the transform rules on the partition id c: If c changes, it changes from 0 to non-zero or vice versa.

Property 3.3 (Temporal Separation) *For all states s in S, for all i, $1 \le i \le n$, if the partition id c_s is 0, then the k data areas of partition i are clear, i.e., $D^1_{i,s} = 0, \ldots, D^k_{i,s} = 0$.*

3.2.4 Separation of Control Property

This property states that when data processing is in progress on partition i, no data is being processed on partition j, $j \neq i$, until processing on partition i terminates. The property is defined in terms of the partition id c, which is i if processing is in progress on partition i, where $i > 0$, and 0 otherwise, and the set D_i of k data areas in partition i, $D_i = \{D^j_i \mid 1 \le j \le k\}$.

Property 3.4 (Separation of Control) *Suppose that states s and s' are in S, event e is in H, data area a is in $\mathcal{M}$, and j, where $1 \le j \le n$, is a partition id. Suppose further that $s' = T(e, s)$. If neither c_s nor $c_{s'}$ is j, then $a_s = a_{s'}$ for all $a \in D_j$.*

3.2.5 Kernel Integrity Property

The Kernel Integrity Property states that when data processing is in progress on partition i, the data stored on memory area G does not change. This property is defined in terms of G and the set P_i of events for partition i.

Property 3.5 (Kernel Integrity) *Suppose that states s and s' are in state set S, event e is in H, and i is a partition id, $1 \le i \le n$. Suppose further that $s' = T(e, s)$. If e is a partition event in P_i, then $G_{s'} = G_s$.*

3.3 Formal Verification

To formally verify that the TLS enforces data separation, the natural language formulation of the TLS was translated into TAME (Timed Automata Modeling Environment) [8, 7], a front-end to the mechanical prover PVS [27] which helps a user specify and reason formally about automata models. This translation requires the completion of a template to define the initial states, state transitions, input events, and other attributes of the state machine Σ. The TAME specification provides a machine version of the TLS that can be shown mechanically to satisfy the five subproperties defined above.

After constructing the TAME specification of the TLS, we formulated two sets of TLS properties in TAME—invariant properties and other properties—that together formalize the five subproperties. Then, for each set of properties, we interactively constructed (TAME) proofs showing that the TAME specification satisfies each property. The scripts of these proofs, which are saved by PVS, can be rerun easily by the evaluators and serve as the formal proofs of data separation. One benefit of TAME is that the saved PVS proof scripts can be largely understood without rerunning them in PVS.

3.4 Partitioning the Code

To show formally that the ED separation kernel enforces data separation, we must prove that the kernel is a secure partial instantiation of the state machine Σ defined by the TLS. The formal verification described in Section 3.3 establishes formally that a strict instantiation of the TLS enforces data separation. A *partial instantiation* of the TLS is an implementation that contains fine-grain details which do not correspond to the state machine Σ defined in the TLS. A *secure partial instantiation* of the TLS is a partial instantiation of the TLS in which the fine-grain details that do not correspond to the TLS are benign. Section 4 contains the formal foundation for the proof that the code is a secure partial instantiation of the TLS.

The proof that the code for the ED kernel is a secure partial instantiation of the TLS is based on a demonstration that all kernel code falls into three major categories and one subcategory, with proofs that the code in each category satisfies certain properties. The categories are as follows:

1. *Event Code* is kernel code which implements a TLS internal event e in H and touches one or more MAIs. For each segment of Event Code, it is checked that
   (i) the concrete translation of the precondition in the TLS for the corresponding event e is satisfied at the point in the kernel code where the execution of the Event Code is initiated, and
   (ii) the concrete translation of the postcondition in the TLS for the corresponding event e is satisfied at the conclusion of Event Code execution.
2. *Trusted Code* is kernel code which touches MAIs but is not Event Code. This code does not correspond to behavior defined by the TLS and may have read and write access both to MAIs and to memory areas outside of the MAIs. It is validated either by a proof that the code does not permit any non-secure information flows or, in rare instances, by assumption. The TLS makes explicit any assumptions used in connection with Trusted Code and its behavior. The proofs for a given segment of Trusted Code characterize the entire functional behavior of that Trusted Code using Floyd-Hoare style assertions at the code level and show that no non-secure information flows can occur in that code.
3. *Other Code* is kernel code that is neither Event nor Trusted Code. More specifically, Other Code is kernel code which does not correspond to any behavior defined by the TLS and which has no access to any MAIs. Apple's Xcode development tool [2] was used to search the kernel code to locate all code segments with access to MAIs, i.e, code segments classified as Event or Trusted Code. This involved identifying all places in the kernel code where the MMU is reset and observing the permissions assigned. By observing the access granted for code segments categorized as Other Code, we can ensure that they have no access to any MAI.
   (a) A subset of Other Code, called *Verified Code*, is code with no access to MAIs which is still security-relevant because it performs functions necessary for the kernel to enforce data separation. These functions include setting up the MMU, establishing preconditions for Event Code, etc. Floyd-Hoare style assertions at the code level are used to prove that Verified Code correctly implements the required functions.

3.5 Demonstrating Code Conformance

Demonstrating code conformance requires the definition of two mappings. To establish correspondence between concrete states in the kernel code and abstract states in the TLS, a function α is defined that relates concrete states to abstract states by relating concrete entities (such as memory areas, code variables, and logical variables) at the code level to abstract state variables (such as MAIs and the partition id) in the TLS. For example, the actual physical addresses of the MAIs are mapped to their corresponding

$\{\texttt{CopyDIn\_partition\_id} : \texttt{partition} = \texttt{partition\_id}\}$
$\{\texttt{CopyDIn\_priv} :$
$\quad \{(\texttt{R}, \texttt{KER\_INBUFFER\_1\_partition}), (\texttt{W}, \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition})\} \subseteq \texttt{MMU}\}$
$\{\texttt{CopyDIn\_value\_data} : \forall \texttt{j}.0 \leq \texttt{j} < \texttt{byte\_length}.$
$\quad \texttt{A}[\texttt{j}] = \texttt{KER\_INBUFFER\_1\_partition\_START} + \texttt{j}\}$
$\{\texttt{CopyDIn\_def\_value\_rest} : \forall \texttt{j}.\texttt{byte\_length} \leq \texttt{j} < \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_SIZE}.$
$\quad \texttt{B}[\texttt{j}] = \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_START} + \texttt{j}\}$
$\{\texttt{CopyDIn\_local\_inbuffer} : \texttt{buffer\_in\_start} = \texttt{KER\_INBUFFER\_1\_partition\_START}\}$
$\{\texttt{CopyDIn\_local\_datain} : \texttt{part\_data\_start} = \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_START}\}$

```
      if (byte_length < (unsigned long)&__INBUFFER_SIZE)
{
      /* copy data from inbuffer 1 to partition      */
      /* part_data_start contains the starting address of */
      /* the memory area, buffer_in_start contains */
      /* the starting address of the inbuffer     */
      /* kernel_memcopy is a copy routine whose functional correctness */
      /* has been verified using Floyd-Hoare assertions */
      kernel_memcopy(part_data_start, buffer_in_start, byte_length);
}
```

$\{\texttt{CopyDIn\_copy\_size\_datain} : \texttt{byte\_length} > \texttt{KER\_PAR\_DATA\_STORAGE\_partition\_SIZE} \rightarrow \mathit{false}\}$
$\{\texttt{CopyDIn\_copy\_size\_inbuffer} : \texttt{byte\_length} > \texttt{KER\_INBUFFER\_1\_partition\_SIZE} \rightarrow \mathit{false}\}$
$\{\texttt{CopyDIn\_gamma\_copy} : \forall \texttt{j}.0 \leq \texttt{j} < \texttt{byte\_length}.$
$\quad \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_START} + \texttt{j} = \texttt{A}[\texttt{j}]\}$
$\{\texttt{CopyDIn\_gamma\_rest} : \forall \texttt{j}.\texttt{byte\_length} \leq \texttt{j} < \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_SIZE}.$
$\quad \texttt{KER\_PAR\_DATA\_STORAGE\_1\_partition\_START} + \texttt{j} = \texttt{B}[\texttt{j}]\}$
$\{\texttt{CopyDIn\_sanitize} : \texttt{part\_data\_sanitized\_partition} = \texttt{false}\}$
$\{$`CopyDIn_NOC` : No concrete state variables have changed value except possibly
$\quad$`KER_PAR_DATA_STORAGE_partition` and `part_data_sanitized_partition`. $\}$

Figure 1. Event Code and Code Level Assertions for Event Copy_Bfr1In_Data1In_i

abstract state variables in the TLS. The map α also maps Event Code to events in the TLS. Another map Φ relates assertions at the abstract TLS level to assertions at the code level derived from the map α. See Section 4 for more details.

Using Φ to relate pre- and postconditions for an event in the TLS to derived pre- and postconditions for the corresponding Event Code, we next determine for each piece of Event Code sets of code-level pre- and postconditions that match the derived pre- and postconditions as closely as possible. Figure 1 shows the Event Code corresponding to the `Copy_Bfr1In_Data1In_i` event in the TLS and the code level pre- and postconditions for this Event Code. In Figure 1, the top box contains the preconditions, then the indented Event Code is listed, and finally the bottom box contains the postconditions; each pre- and postcondition has the form $\{\texttt{Assertion\_Name} : \texttt{Assertion}\}$. Generally, the match between assertions in the TLS and derived code-level assertions is not exact because auxiliary assertions are added 1) to express the correspondence between variables in the code and physical memory areas[3] (e.g., `CopyDIn_local_datain`), 2) to save values in memory areas as the values of logical variables (e.g., `CopyDIn_value_data`), and 3) to express error conditions that the TLS implicitly assumes to be impossible (e.g., `CopyDIn_copy_size_datain`).

After defining the desired sets of code-level pre- and postconditions, we check whether these assertions are among the assertions already proven in the annotated C code. The annotated C code refers to memory areas by indexing into arrays that define memory maps in the code, whereas the mapping α refers to memory areas by their actual physical addresses. Thus, to be equivalent to the desired assertions, the assertions in the annotated code frequently need dereferencing. For example, the annotated C code assertion §8.4, TLS2, is defined by

$$\texttt{part\_data\_start} =$$
$$(\texttt{unsignedchar}*)\texttt{ker\_rtime\_mmu\_map}[\texttt{partition}].\texttt{part\_data\_start},$$

which sets the variable `part_data_start` to the starting address of the data area in the partition by indexing into the real-time memory map in the code and selecting the `part_data_start` member of the structure corresponding to that array element. Dereferencing the index into the array and pointer into the structure yields the memory area `KER_PAR_DATA_STORAGE_1_partition_START`, the actual physical address of the partition data area, which stores the value used in the code-level precondition `CopyDIn_local_datain`.

In our initial attempt to match a pre- and postcondition in the annotated C code with each desired pre- and postcondition, four different outcomes were possible:

- The desired assertion exactly matched an assertion in the annotated code.
- The desired assertion exactly matched an assertion in the annotated code but dereferencing was required.
- The desired assertion was a close match with an assertion in the annotated code.
- No code assertion exactly or approximately matched the desired assertion.

We worked with the group annotating the C code to ensure that assertions corresponding to all desired pre- and postconditions were added to and verified on the code. (In general, it is sufficient to include strongest postconditions implying our derived assertions.) To show correspondence between the pre- and postconditions in the code and the TLS, two tables were created for each TLS event.

---

[3] This facilitates Floyd-Hoare reasoning at the code level.

Table 1. Mapping Preconditions in the Code to Preconditions in the TLS

Precondition $\Phi(\texttt{pre}_e)(s_c)$ Desired in the Code	Assertion in Annotated Code	Precondition $\texttt{pre}_e(s)$ in the TLS	Ref. No.	Description
`CopyDIn_partition_id`	§8.4,P5	$c_s = i$	(1)	Partition id is i
`CopyDIn_priv`	§8.4,TLS1*	$\mathrm{AM}(e, B_i^1) = \texttt{R}$ $\mathrm{AM}(e, D_i^1) = \texttt{W}$	(2)	`R` access for Input Buffer 1, `W` access for Data Area 1
`CopyDIn_value_data`	§8.4,P4*	$B_{i,s}^1$	-	Value of data in Input Buffer 1
`CopyDIn_def_value_rest`	§8.4,TLS4	$D_{i,s}^1$	-	Value of Data Area 1
`CopyDIn_local_inbuffer`	§8.4, TLS3*	-	-	Local variable for Input Buffer 1
`CopyDIn_local_datain`	§8.4,TLS2*	-	-	Local variable for Data Area 1

Table 2. Mapping Postconditions in the Code to Postconditions in the TLS

Postcondition $\Phi(\texttt{post}_e)(s_c, s_c')$ Desired in the Code	Assertion in Annotated Code	Postcondition $\texttt{post}_e(s, s')$ in the TLS	Ref. No.	Description
`CopyDIn_copy_size_datain`	§8.4,R2*	-	-	Wrong size → Error return
`CopyDIn_copy_size_inbuffer`	§8.4, R3*	-	-	Wrong size → Error return
`CopyDIn_gamma_copy`	§8.4, R7*	$D_{i,s'}^1 = \Gamma(B_{i,s}^1)$	(4)	Copy to Data Area 1
`CopyDIn_gamma_rest`	§8.4,TLS6	-		Rem Data Area 1 unchged
`CopyDIn_sanitize`	§8.4,TLS5*	$\mathcal{W}_{D,s'}^1[i] = false$	(3)	Data Area 1 not sanitized
`CopyDIn_NOC`	By inspection	$\mathrm{NOC}_{\{\mathcal{W}_D^1[i], D_i^1\}}$	(5)	No other change

Tables 1 and 2 are the correspondence tables for the pre- and postconditions for the TLS event $e = \texttt{Copy\_Bfr1In\_Data1In\_}i$ defined in Section 3.1. In the tables, s and $s' = T(e, s)$ represent the abstract pre- and poststate; s_c, and s_c' represent the concrete pre- and poststate; and Φ, which is formally defined in Section 4, maps abstract predicates to corresponding concrete predicates.

In the tables, the first column contains the label of a desired code-level pre- or postcondition, the second column gives the location (section number and assertion label) of the corresponding assertion in the annotated C code, the third column contains the corresponding pre- or postcondition (if any) in the TLS, the fourth column gives the reference number of the corresponding assertion in the transform rule, and the fifth column briefly describes the assertion. In cases where no corresponding assertion exists in the TLS, '-' appears in both the third and fourth columns. An asterisk in the second column indicates that, for equivalence between the assertion in the annotated code and the desired code assertion to hold, the assertion in the annotated code requires dereferencing.

4. Formal Foundations

This section formalizes our method for showing that the kernel code conforms to the behavior captured in the TLS. To begin, a function α is defined that maps each concrete state at the code level to a corresponding abstract state in the TLS state machine Σ by relating variables at the concrete code level to variables at the abstract TLS level. Variables at the concrete level include variables in the code, predicates defined on the code, logical history variables, and memory areas. Among the most important memory areas treated as concrete state variables are the data areas and the input and output buffers assigned to each partition, which are central to reasoning about possible information flows. Because each possible value of a concrete state variable can be represented by some possible value of the corresponding abstract state variable, the map α from concrete to abstract state variables induces a map $\alpha : S_\mathrm{c} \rightarrow S_\mathrm{a}$ from concrete to abstract states in the obvious way.[4]

Once α is defined at the level of states in terms of state variables, the set $\mathrm{E_c}$ of Event Code code segments transferring data either to or from an MAI in the current partition is identified, and α is extended to map each code segment $\mathrm{e_c}$ in $\mathrm{E_c}$ to a corresponding internal event $\mathrm{e_a} = \alpha(\mathrm{e_c})$ in the TLS.

The map α from concrete states to abstract states provides a means to take any assertion P_a about abstract states and derive a corresponding assertion $\Phi(\mathrm{P}_a)$ about concrete states as follows:

$$\Phi(\mathrm{P}_a)(s_\mathrm{c}) \triangleq \mathrm{P}_a(\alpha(s_\mathrm{c})),$$

where s_c is any state in S_c. Analogously, α can be used to derive an assertion $\Phi(\mathrm{P}_a)(s_\mathrm{c}^1, s_\mathrm{c}^2)$ about a pair of concrete states from an assertion about a pair of abstract states as follows:

$$\Phi(\mathrm{P_a})(s_\mathrm{c}^1, s_\mathrm{c}^2) \triangleq \mathrm{P_a}(\alpha(s_\mathrm{c}^1), \alpha(s_\mathrm{c}^2)).$$

The map Φ is used to relate preconditions and postconditions in the code to preconditions and postconditions in the TLS (see Figure 2). Note that preconditions (at both levels) apply only to one state. To capture the fact that an event changes only certain state variables (indicated at the abstract level using `NOC`), the postconditions are represented at both levels as predicates on two states.

To establish equivalence between the behavior of the kernel code and a subset of the behavior modeled in the TLS, it is sufficient to prove, in the simplest case, that for every $\mathrm{e_c}$ in $\mathrm{E_c}$,

1. Whenever the concrete code segment e_c is ready to execute in state s_c, some concrete precondition $\mathrm{Pre}_{\mathrm{e_c}}$ holds, where $\mathrm{Pre}_{\mathrm{e_c}}$ implies $\Phi(\mathrm{Pre}_{\mathrm{e_a}})$, the concrete precondition derived from the abstract precondition for $\mathrm{e_a} = \alpha(e_\mathrm{c})$;

2. Whenever the concrete precondition $\mathrm{Pre}_{\mathrm{e_c}}$ holds for the current program state $\mathrm{s_c}$, some concrete postcondition $\mathrm{Post}_{\mathrm{e_c}}$ holds for the pair of program states (s_c,$e_\mathrm{c}(s_\mathrm{c})$) immediately before and immediately after execution of e_c, where $\mathrm{Post}_{\mathrm{e_c}}$ implies $\Phi(\mathrm{Post}_{\mathrm{e_a}})$, the concrete postcondition derived from the abstract postcondition for e_a;

3. The diagram in Figure 2 commutes when conditions 1 and 2 are satisfied and $\mathrm{Pre}_{\mathrm{e_c}}(s_\mathrm{c})$ holds.

---

[4] To distinguish abstract from concrete entities, this section tags abstract entities with an a and concrete entities with a c; for example, S_a represents the abstract states s and S_c the concrete states s.

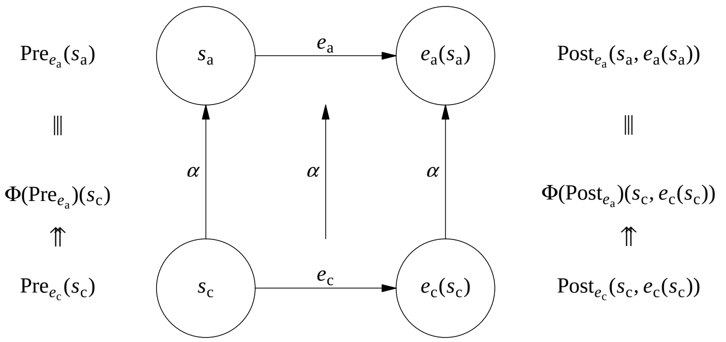
$\mathrm{Pre}_{e_a}(s_a)$
s_a
e_a
$e_a(s_a)$
$\mathrm{Post}_{e_a}(s_a, e_a(s_a))$
$\equiv$
$\equiv$
α
α
α
$\Phi(\mathrm{Pre}_{e_a})(s_c)$
$\Phi(\mathrm{Post}_{e_a})(s_c, e_c(s_c))$
$\Uparrow$
$\Uparrow$
$\mathrm{Pre}_{e_c}(s_c)$
s_c
e_c
$e_c(s_c)$
$\mathrm{Post}_{e_c}(s_c, e_c(s_c))$

Figure 2. Relation between concrete and abstract transitions.

Provided $\mathrm{Post}_{e_a}(s_a, s'_a) \equiv (s'_a = e_a(s_a))$ (as holds for `post_e` in the TLS transform described in Section 3.1), to establish condition 3, it is sufficient to prove that $\mathrm{Pre}_{e_c}(s_c) \Rightarrow \Phi(\mathrm{Pre}_{e_a})(s_c)$ and that $\mathrm{Post}_{e_c}(s_c, e_c(s_c)) \Rightarrow \Phi(\mathrm{Post}_{e_a})(s_c, e_c(s_c))$. Establishing conditions 1– 3 guarantees that whenever the code segment e_c executes in the code, there is an enabled event e_a in the TLS that causes a transition from the abstract image s_a under α of the concrete prestate s_c at the code level into an abstract state $e_a(s_a)$ that is the abstract image under α of the concrete poststate $e_c(s_c)$ at the code level. More concisely, conditions 1, 2, and 3 imply that there exists an abstract transition that models the concrete transition.

The relation of Event Code segments to abstract events can be slightly more complex than shown in Figure 2. For example, in some cases, e_c may implement more than one event. However, these more complex cases can be handled similarly. When a concrete event implements n abstract events, for example, one looks for a partition $\mathrm{Pre}_c \equiv \mathrm{Pre}_c^1 \oplus ... \oplus \mathrm{Pre}_c^n$ of the concrete precondition Pre_c such that when the ith part Pre_c^i holds, the code e_c implements the ith abstract event. Then, one establishes for each i a commutative diagram analogous to the diagram in Figure 2.

The argument that the kernel ensures data separation is based on relating executions of the kernel code to executions in the TLS. It begins by observing that α maps ED's initial state via α to an allowed initial state in the TLS. To support the remainder of the argument, the Event Code set E_c and the code-level map α are extended to cover the Other Code, and it is shown that the Trusted Code can be safely ignored. Most Event Code segments consist of a single program statement. In contrast, Other Code contains many lengthy code segments which simply manipulate local variables inside a function or procedure and do not map to any abstract event; such segments typically occur prior to an Event Code segment. We model these Other Code segments at the abstract level by a `no−op` ("do nothing") event implicitly included in the TLS.

Because every code segment in the Event or Other Code is modeled either by an abstract TLS event with concrete and abstract transitions related as in Figure 2 or by a `no−op` in the TLS, it follows that every execution of this part of the code corresponds to an execution in the TLS. Because parts of the Trusted Code have been verified and the remaining parts have been certified to cause no insecure information flows, modeling this code at the abstract level is unnecessary. Combining this reasoning with the additional assurance that α relates concrete data and buffer memory areas to abstract ones and thus models all information flows involving Memory Areas of Interest, it follows that all kernel behavior relevant to data separation at the concrete level is modeled at the abstract level. Thus, the Data Separation Property proved at the abstract level also holds at the concrete level.

5. Lessons Learned

5.1 Software Design Decisions

Three software design decisions were critical in making code verification feasible. One major decision was to use a separation kernel, a single software module to mediate all memory accesses. A design that distributed the checking of memory accesses would have made the task of proving data separation much more difficult, if not impossible. A second critical decision was to keep the software simple. For example, once initiated, data processing in a partition was run to completion unless an exception occurred. In addition, ED's services were limited to the essential ones—the temptation to add new services late in development was resisted. A third critical decision was to enforce the "least privilege principle." For example, if a process only required read access to a memory area, the kernel only granted read, and not write, access.

5.2 Top-Level Specification

One major challenge was to understand the required behavior of the separation kernel. Both scenarios and the SCR tools [19, 18] were useful in validating and extending our understanding of the kernel behavior. To begin, we formulated several scenarios, i.e., sequences of input events and how the kernel responded to those events. After specifying a state machine model of the kernel in SCR, we ran the scenarios through the SCR simulator. As expected, formulating the scenarios and running them through the simulator exposed gaps in our understanding. Both the scenarios and the questions raised were valuable in eliciting details of the required kernel behavior from ED's development team.

Keeping the size of the TLS small was critical for many reasons. It simplified communicating with the other stakeholders, changing the specification when the kernel behavior changed, translating the specification into TAME, and proving that the TLS enforced data separation.

The natural language representation of the TLS enabled stakeholders from differing backgrounds and with different objectives—the project manager, the software developers, and the evaluators—to communicate easily with the formal methods team about the kernel's required behavior. Discussion among these various stakeholders helped ensure that misunderstandings were avoided and issues resolved early in the certification process. This natural language representation of the TLS for ED contrasts with the representations used in many other formal specifications of secure systems, which are often expressed in specialized languages such as ACL2 (see, e.g, [17]). Moreover, any ambiguity inherent in the natural language representation was removed by translating the TLS into TAME, since the state machine semantics underlying TAME is expressed as a PVS theory.

5.3 Mechanized Verification

TAME's specification and proof support significantly simplified the verification effort. Translating the TLS into TAME required about three days. Because the number of memory areas is unspecified in the TLS, the overall memory content in the TLS had to be captured in TAME as a function from a set of memory areas to storable values. The higher order nature of PVS, which made this feasible, also contributed to the compactness of the TAME specification, which is only 368 lines long. In translating the TLS to TAME, the correspondence between entities in the natural language formulation and TAME entities was documented. Adjusting the TAME specification to reflect later changes in the TLS required less than three hours. Representing the five subproperties in TAME required about two hours.

About two weeks were needed to formally verify that the TLS enforces data separation. Adding and proving a new property (Kernel Integrity) suggested by an evaluator required under one hour. In proving the subproperties, a few days were needed to formulate an efficient proof approach. This exploration led to a new PVS strategy designed to simplify the proof guidance in the most complex proof. This strategy was useful in proving all five subproperties and has also been useful in other TAME applications. The proof of each subproperty completes in two minutes or less. Once the correct proof approach was identified, the time required to develop the proof scripts interactively in TAME was one day.

5.4 Showing Code Conformance

Two months were required to establish conformance between the TLS and the annotated C code. In the first month, we experimented with several different approaches for demonstrating conformance before the approach presented in this paper was selected. Once an approach was selected, the formal correspondence argument required one week. Three weeks were needed to construct the correspondence of Event Code to TLS events, i.e., developing the code level assertions necessary for the TLS pre- and postconditions to hold and locating the corresponding assertions in the annotated C code. One day was spent using the Xcode tool to locate all Event and Trusted Code and to verify that the permissions for the Other Code did not include access to MAIs. One week was needed to add the required assertions to the annotated code.

Our method for demonstrating code conformance relies on the notions of MAIs and Event Code. The extent to which our method can be extended to other applications depends on whether an analogous method of identifying Event Code (and Trusted Code) can be found. This is likely to be possible in other applications that must enforce data separation.

6. Open Problems

6.1 Checking and Constructing Code Annotations

For many years, researchers have recommended annotating code with pre- and postconditions and invariants (see, e.g., [25]). Code annotations are already used in practice. For example, software developers at Praxis annotate Spark programs with assertions and use tools to automatically check the validity of the assertions [10]. Moreover, at Microsoft, annotations are a mandated part of the software development process in the largest product groups [12]. However, manual annotation of source code with pre- and postconditions remains rare in the wider software development community because it is both tedious and error-prone. Hence, automated tools for checking code annotations would be extremely valuable. Even more valuable are tools that can construct pre- and postconditions automatically. One approach may be for a developer to generate some key pre- and postconditions. Given a small set of annotations, a tool could then generate additional annotations automatically.

6.2 A Code Conformance Proof Assistant

The semantic distance between the abstract TLS required for a Common Criteria evaluation and a low-level C program is huge. While the TLS describes the security-relevant program behavior in terms of sets, functions, and relations, the description of the behavior of a C program is in terms of low-level constructs, such as arrays, integers, and bits stored in registers and memory areas. Hence, automatic demonstration of conformance of low level C code to a TLS is unrealistic. A more realistic goal is a proof assistant with two inputs, a C program annotated with assertions and a TLS of the security-relevant functions of that program, for helping the user establish that the C program satisfies the TLS.

6.3 Automatic Code Generation

One promising way to obtain high assurance that an implementation satisfies critical security properties is to generate code automatically from a specification that has been proven to satisfy the properties. Automatic code generation is already feasible for some low-level specification languages such as Esterel [1]. While constructing efficient source code from more abstract specifications is possible for simple program constructs using simple data types (see, e.g., [30]), new research is needed to produce efficient code from specifications containing richer constructs and data types. Such technology should drastically reduce the effort required to produce efficient code and to increase assurance that the code satisfies critical security properties.

7. Related Work

In the 1980s, the SCOMP [13], SeaView [22], LOCK [35], and Multinet Gateway [14] projects all applied formal methods to the specification and verification of systems. All developed TLSs and formal statements of the system security policies. For SCOMP, Multinet Gateway, and LOCK, the TLS was shown formally to satisfy the security policy. For SeaView, only two of 31 operations in the TLS were verified against the security policy model [36]. Conformance between the TLS and the SCOMP code was shown by constructing several mappings: English language to TLS, TLS to pseudo-code, and TLS to actual code [11]. The mapping was top down from the TLS to code; as a result, some code was unmapped. This approach is similar to our mapping of Event Code to the TLS, although the mapping is in the other direction. The LOCK project constructed mappings partially relating the TLS to the source code; specification-based testing provided additional evidence of correspondence. In Multinet Gateway, verification conditions were generated to show conformance between the specification and the code. If and how these conditions were discharged is unclear. Each project used tools to aid in specification and verification: SCOMP used HDM [29], Seaview used EHDM [32], and Multinet Gateway and LOCK used Gypsy [15]. More recently, in 2006, we formulated a second possible approach to software verification, based on TAME, which uses verified formal pseudocode as "glue" relating a TLS to actual code [9].

In [17, 5], Greve, Wilding, and Vanfleet (GWV) present an ACL2 model for a generic separation kernel. In the model, a function describes the possible information flows between memory areas. This notion of flow is not as fine-grained as in our model, where access (with its possible information flows) is granted to each process only when it executes in a partition, thus providing least privilege in addition to separation. In the GWV approach, separation includes No-Exfiltration and No-Infiltration but not Temporal Separation, since the model does not allow reconfigurable partitions. How the GWV model was used to verify the AAMP7 microprocessor is described in [16, 28]. A traditional verification process was followed: build a formal security policy, an abstract and detailed model, and an implementation; prove that the abstract model satisfies the security policy; and show correspondence between the abstract and detailed models and between the detailed model and the implementation. Whether correctness was proven at either the detailed design level or code level is unclear.

8. Conclusions

This paper has introduced a novel and affordable approach for verifying security down to the source code level. The approach begins with a well-defined security policy, builds the minimal state machine model needed to prove that the model satisfies the policy, and proves, using a mechanical verifier, that the security model satisfies the policy. Once complete, the code is annotated with preconditions and postconditions and then partitioned into Event, Trusted, and Other Code. The final step is to 1) demonstrate conformance of the Event Code and the code pre- and postconditions with the internal events and pre- and postconditions of the TLS and 2) show that the Trusted Code and the Other Code are benign.

Tools such as model checkers and theorem provers are already available for verifying that a formal specification satisfies a security policy. A research challenge is to develop tools 1) for validating and constructing pre- and postconditions from source code, including C code, 2) to help show conformance of annotated code with a TLS, and 3) to automatically construct efficient, provably correct code from specifications. Research that addresses these three problems should significantly increase the affordability of constructing verified security-critical software.

9. Acknowledgments

We acknowledge the monumental effort of the group who annotated the kernel code with pre- and postconditions and of the ED project leader, who had the foresight to include a separation kernel and to keep the design simple. Without the annotated code and solid design decisions, our effort would have been impossible. We also thank the members of the ED design team for answering questions about ED's operational behavior.

10. REFERENCES

[1] SCADE Tool Suite. Tools and documentation available at http://www.esterel-technologies.com/products/scade-suite.

[2] Xcode 2.1. Tool and documentation available at http://developer.apple.com/tools/xcode/index.html.

[3] Common criteria for information technology security evaluation, Parts 1–3. Tech. Report CCIMB-2004-01-001—003, Version 2.2, Rev. 256, Jan. 2004.

[4] C. Adams. Keeping secrets in integrated avionics. *Aviation Today*, 2004.

[5] J. Alves-Foss and C. Taylor. An analysis of the GWV security policy. In *5th Internat. Workshop on ACL2 Prover and Its Applications (ACL2-2004)*, 2004.

[6] J.P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, MA, 1972.

[7] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Eng.*, 9:201–232, 2002.

[8] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Math. and Artificial Intelligence*, 29(1-4):131–189, 2000.

[9] M. Archer and E. Leonard. Establishing high confidence in code implementations of algorithms using formal verification of pseudocode. In *Proc., 3rd Internat. Verification Workshop (VERIFY '06)*, Seattle, WA, August 2006.

[10] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[11] T.V. Benzel. Analysis of a kernel verification. In *Proceedings of the IEEE Security and Privacy Conference*, April 1984.

[12] M. Das. Formal specifications on industrial-strength code – From myth to reality. In *Proc., Computer-Aided Verification (CAV 2006)*, Seattle, WA, August 2006.

[13] L.J. Fraim. Secure office management system: The first commodity application on a trusted system. In *Proc., 1987 Fall Joint Computer Conf. on Exploring Technology: Today and Tomorrow*, 1987.

[14] S. Gerhart, D. Craigen, and T. Ralston. Case study: Multinet Gateway System. *IEEE Software*, pages 37–39, 1994.

[15] D.I. Good. *Mechanical Proofs about Computer Programs*, chapter 3, pages 55–75. Prentice Hall, 1985.

[16] D. Greve, R. Richards, and M. Wilding. A summary of intrinsic partitioning verification. In *Fifth Internat. Workshop on the ACL2 Prover and Its Applications (ACL2-2004)*, 2004.

[17] D. Greve, M. Wilding, and W.M. Vanfleet. A separation kernel formal security policy. In *Fourth Internat. Workshop on the ACL2 Prover and Its Applications (ACL2-2003)*, July 2003.

[18] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, 1998.

[19] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. on Software Eng. and Methodology*, 5(3):231–261, 1996.

[20] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *Proc., 27th Internat. Conf. on Software Engineering*, St. Louis, MO, 2005.

[21] C. E. Landwehr, C. L. Heitmeyer, and J. McLean. A security model for military message systems. *ACM Trans. on Computer Systems*, 2(3):198–222, August 1984.

[22] T.F. Lunt, D.E. Denning, R.R. Schell, M. Heckman, and W.R. Shockley. The SeaView security model. *IEEE Trans. on Software Eng.*, 16(6), 1990.

[23] J. McLean, C. Landwehr, and C. Heitmeyer. A formal statement of the military message system security model. In *Proc., 1984 IEEE Symposium on Security and Privacy*, pages 188–194, 1984.

[24] C. Meadows. Analysis of the internet key exchange protocol using the NRL protocol analyzer. In *IEEE Symp. on Security and Privacy*, Oakland, 1999.

[25] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10), 1992.

[26] J Strother Moore, Thomas W. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5K86TM floating-point division program. *IEEE Transactions on Computers*, 7(9), 1998.

[27] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Software Eng.*, 21(2), 1995.

[28] R. Richards, D. Greve, M. Wilding, and W.M. Vanfleet. The common criteria, formal methods and ACL2. In *Fifth Internat. Workshop, ACL2 Prover and Its Applications (ACL2-2004)*, 2004.

[29] L. Robinson. The HDM handbook, volume 1: The foundations of HDM, SRI project 4828. Tech. report, SRI International, Menlo Park, CA, June 1979.

[30] T. Rothamel, C. Heitmeyer, E. Leonard, and A. Liu. Generating optimized code from SCR specifications. In *Proc., ACM SIGPLAN/SIGBED Conf. on Languages, Compilers and Tools for Embedded Systems (LCTES 2006)*, Ottawa, Canada, June 2006.

[31] J. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, CA, August 1994.

[32] J. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report CSL-91-2, SRI International, Menlo Park, CA, February 1991.

[33] J. Rushby. Design and verification of secure systems. In *Proc., Eighth ACM Symp. on Operating System Principles*, pages 12–21, Dec. 1981.

[34] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Computer Science Lab, SRI Internat., Menlo Park, CA, November 2001.

[35] R. Smith. Cost profile of a highly assured, secure operating system. *ACM Transactions on Information and System Security*, 4(1):72–101, 2001.

[36] R.A. Whitehurst and T.F. Lunt. The SeaView verification. In *Proc., Computer Security Foundations Workshop II, 1989*, June 1989.